```
Trying 01083...Open

PLEASE ENTER HOST PORT ID:
PLEASE ENTER HOST PORT ID:x
LOGINID:d232dye
PASSWORD:
 * * * * * * RECONNECTED TO U.S. Patent & Trademark Office * * * * * *
SESSION RESUMED IN FILE 'USPAT' AT 16:33:42 ON 02 SEP 1999
FILE 'USPAT' ENTERED AT 16:33:42 ON 02 SEP 1999
=> d his


     (FILE 'USPAT' ENTERED AT 16:13:11 ON 02 SEP 1999)
L1          5147 S 712/?/CCLS
L2            47 S (EXECUTION (W) STACK?)
L3          2779 S L1 AND (EXCEPTION? OR INTERRUPT?)
L4            11 S ((EXECUTION (W) STACK?) (P) (EXCEPTION? OR INTERRUPT?))
L5             1 S L4 AND JAVA?

=> s ((java?) (p) (exception? or interrupt?))

          1161 JAVA?
        154378 EXCEPTION?
        228578 INTERRUPT?
L6          28 ((JAVA?) (P) (EXCEPTION? OR INTERRUPT?))

=> s ((java?) (p) stack? (p) (exception? or interrupt?))

          1161 JAVA?
        172617 STACK?
        154378 EXCEPTION?
        228578 INTERRUPT?
L7           6 ((JAVA?) (P) STACK? (P) (EXCEPTION? OR INTERRUPT?))

=> d l7 kwic 1-6


US PAT NO:     5,937,193 [IMAGE AVAILABLE]          L7: 1 of 6

DETDESC:

DETD(54)
```

Returning to block 210, translation state machine 153 handles **exception** conditions by testing for an EXC code. Upon receipt of this code, control is passed to block 212 where the translation process is halted, the REQ signal is released, and an **interrupt** is sent to processor 140. The **exception** condition is generally implemented when a platform-independent instruction cannot or will not be translated by translation circuit 150, thereby enabling processor 140 to interpret the instruction. An example of a **Java** bytecode that may generate an **exception** is the "ldcl" bytecode (12h), which pushes an item, which may have varying lengths (e.g., a string), onto the **stack**. The translation circuit shown herein does not include any mechanism to determine the length of an item, and consequently, processor. . .

```
US PAT NO:     5,925,123 [IMAGE AVAILABLE]          L7: 2 of 6

DETDESC:
```

DETD(22)

One . . . that are executed directly by hardware processor 100 is described herein by way of an example. Thirty percent of the **JAVA** virtual machine instructions are pure hardware translations; instructions implemented in this manner include constant loading and simple **stack** operations. The next 50% of the virtual machine instructions are implemented mostly, but not entirely, in hardware and require some firmware assistance; these include **stack** based operations and array instructions. The next 10% of the **JAVA** virtual machine instructions are implemented in hardware, but require significant firmware support as well; these include function invocation and function return. The remaining 10% of the **JAVA** virtual machine instructions are not supported in hardware, but rather are supported by a firmware trap and/or microcode; these include functions such as **exception** handlers. Herein, firmware means microcode stored in ROM that when executed controls the operations of hardware processor 100.

DETDESC:

DETD(157)

The **JAVA** Virtual Machine Specification defines that certain instructions can cause certain **exceptions**. The checks for these **exception** conditions are implemented, and a hardware/software mechanism for dealing with them is provided in hardware processor 100 by information in. . . The alternatives include having a trap vector style or a single trap target and pushing the trap type on the **stack** so that the dedicated trap handler routine determines the appropriate action.

US PAT NO: 5,923,892 [IMAGE AVAILABLE] L7: 3 of 6

DETDESC:

DETD(34)

If the currently-processed instruction is not an **exception** instruction, control passes from block 118 to block 120 where the instruction is processed in accordance with the **Java** Virtual Machine Specification. Next, in block 122 a **stack** underflow condition is tested, which occurs when the task or routine allocated to the coprocessor 38 has been completed and. . . where the task complete flag in block 58 is set. Coprocessor interface 54 also includes dedicated logic to signal an **interrupt** to host processor 22. Alternatively, in another embodiment the **interrupt** is positively asserted in block 124.

US PAT NO: 5,884,083 [IMAGE AVAILABLE] L7: 4 of 6

DETDESC:

DETD(18)

In . . . LALR(1) parser are generated for input to a Smalltalk.TM. compiler are provided. The production rules are substantially taken from the **Java**.RTM. language specification, although **exceptions** should be noted below. Following the production rule is a schematic for the resulting parse node which, as will be. . . is assumed, for purposes of the following description, that the parse node for a production rule is left on a **stack** which is maintained by the parser. The notation used here utilizes names from the production rules to denote parse nodes that are left on the **stack** as part of the parsing process. Where alternatives are noted, such as NamelPrimaryNoNewArray, this is an abbreviated way of denoting. . . depending upon which part of the

production rule was actually parsed, thereby determining what parse node was left on the **stack**.

US PAT NO: 5,784,553 [IMAGE AVAILABLE] L7: 5 of 6

DETDESC:

DETD(163)

FIG. 17 is a functional block diagram of the TGS Driver Program for operating on **JAVA** bytecodes. The Symbolic Virtual Machine (SVM) 234 performs both a normal and a symbolic execution of the **JAVA** program represented by the **JAVA** bytecodes 210. In this embodiment, the **JAVA** bytecodes replace the instrumented object code 25 and the SVM 234 replaces the Execution Module 40 and Symbolic Execution Module. . . FIG. 5. The SVM 234 reads the .class files and performs normal and symbolic execution by having both normal (numeric) **stacks** and symbolic **stacks** in the virtual machine. Interfaces to the **JAVA** Class Libraries 218 specify what symbolic values correspond to specific calls to the library routines. That is, the library interfaces. . . input parameters and how the input is used. The symbolic values are used for both forcing all branches in the **JAVA** program so that as much as possible of the program is covered, and for looking for **exceptions** that can happen at each step in the execution of the program and checking if there are input values that will cause those **exceptions**. If such input values are found, the user is notified that an **exception** can occur at a specific point in the program and what inputs will generate the **exception**. Input can includes all forms of input data, such as text input, graphical input, network input, etc.

DETDESC:

DETD(191)

TABLE IV

```
$tgs Test
-------------------------------------------------------
--> EXCEPTION "ArrayIndexOutOfBoundsException" possible
- - - - - - - - - - - - - - - - - - - - - - - - - - - -
Stack trace:
 <Method Test.main([Ljava/lang/String;) V>, pc = 36
  [./Test.java, line 15]
For inputs:
S0 = [148:+INF], from <Method nextToken>
 called at: [Test.java,
 line 8]
```

DETDESC:

DETD(203)

```
 . . .
tgs -module MT.sub.-- Basic
--> testing <Method MT.sub.-- Basic.method1 ()V>
NO ERRORS FOUND
--> testing <Method MT.sub.-- Basic.method2 ()V>
-----------------------------------------------------
-->EXCEPTION "ArrayIndexOutOfBoundsException" possible
- - - - - - - - - - - - - - - - - - - - - - - - - - -
Stack trace:
 <Method MT.sub.-- Basic.method2 (I) V>, pc = 8
  [./MT.sub.-- Basic.java, line 11]
- - - - - - - - - - - - - - - - - - - - - -. . .
```

DETDESC:

DETD(107)

In the **Java** bytecode language, the "finally" construct is implemented using the **exception** handling facilities, together with a "jsr" (jump to subroutine) instruction and "ret" (return from subroutine) instruction. The cleanup code is implemented as a subroutine. When it is called, the top item on the **stack** will be the return address; this return address is saved in a register. A "ret" is placed at the end. .

.

DETDESC:

DETD(130)

TABLE 1

BYTECODES IN **JAVA** LANGUAGE

INSTRUCTION NAME	SHORT DESCRIPTION
nop	no operation
aconst.sub.-- null	push null object
iconst.sub.-- m1	push. . . dup top 2 elements. Skip one
dup2.sub.-- x2	dup top 2 elements. Skip two
swap	swap top two elements of **stack**.
iadd	integer add
ladd	long add
fadd	floating add
dadd	double float add
isub	integer subtract
lsub	long subtract
fsub. . .	
ifgt	goto if greater than
ifle	goto if less than or equal
if.sub.-- icmpeq	compare top two elements of **stack**
if.sub.-- icmpne	compare top two elements of **stack**
if.sub.-- icmplt	compare top two elements of **stack**
if.sub.-- icmpge	compare top two elements of **stack**
if.sub.-- icmpgt	compare top two elements of **stack**
if.sub.-- icmple	compare top two elements of **stack**
if.sub.-- acmpeq	compare top two objects of **stack**
if.sub.-- acmpne	compare top two objects of **stack**
goto	unconditional goto
jsr	jump subroutine
ret	return from subroutine
tableswitch	goto (case)
lookupswitch	goto (case)
ireturn	return integer from. . . a new array of non-objects
anewarray	Create a new array of objects
arraylength	get length of array
athrow	throw an **exception**

```
checkcast        error if object not of given type
instanceof       is object of given type?
monitorenter     enter a monitored region of. . .
```

DETDESC:

DETD(131)

TABLE 2

Pseudocode for **JAVA** Bytecode Verifier

```
Receive Object Class File with one or more bytecode programs to
be verified.
/*  Perform initial checks. . . the constant pool
      does not match the data type of the referenced constant pool
      item,
      (E) any exception handler does not have properly specified
      starting and ending points,
      {
      Print appropriate error message
      Return with Abort return code
      }
Create: status data structures: stack counter, stack status
array,
register status array, jsr bit vector array
Create SnapShot array with one SnapShot for every instruction in the
bytecode program
Initialize SnapShot for first instruction of program to indicate the
stack is empty and the registers are empty except for data types
indicated by the method's type signature (i.e., for arguments. . .
(e.g., in sequential order in program)
whose changed bit is set
Load SnapShot for the selected instruction (showing status of
stack and registers prior to execution of the selected
instruction) into the stack counter, virtual stack and the
virtual register array, and jsr bit vector array, respectively.
Turn off the selected instruction's changed bit
/*      Emulate the effect of this instruction on the stack and
        registers*/
Case(Instruction Type):
{
Case=Instruction pops data from Operand Stack
        {
        Pop operand data type information from Virtual Stack
Update Stack Counter
If Virtual Stack has Underflowed
        {
        Print error message identifying place in program that
          underflow occurred
        Abort Verification
        Return with abort return code
        }
Compare data type of each operand popped from virtual
        stack with data type required (if any) by the bytecode
        instruction
If type mismatch
        {
        Print message identifying. . . occurred
        Set VerificationSuccess to False
        Return with abort return code
        }
}
Case=Instruction pushes data onto Operand Stack
{
```

```
Push data type information onto Virtual Stack
Update stack counter
If Virtual Stack has Overflowed
        {
        Print message identifying place in program that
          overflow occurred
        Set VerificationSuccess to False.   .   .  new data type
If instruction places an uninitialized object in a register and
        the instruction is protected by any exception handler
        (including the special exception handler for a "finally"
        code block)
        {
        Print error message
        Set VerificationSuccess to False
        }
}
Case=Backwards Branch
{
If Virtual Stack or Virtual Register Array contain any
        uninitialized object data types
        {
        Print error message
        Set VerificationSuccess.   .   .  an
        unconditional goto, a return, or a throw,
(B)     the target of a conditional or unconditional branch,
(C)     all exception handlers for this instruction,
(D)     when the current instruction is a return instruction, the
        successor instructions are the instructions.   .   .  "fall off"
the last instruction
{
Set VerificationSuccess to False
Return with Abort return code value
}
/*  Merge the stack counter, virtual stack, virtual register
array and
    jsr bit
vector arrays into the SnapShots of each of the successor
instructions */
Do for each successor instruction:
{
If the successor instruction is the first instruction of an exception
handler,
{
Change the Stack Status portion of the SnapShot of the
        successor instruction to contain a single object of the
        exception type indicated by the exception handler
        information.
Set stack counter of the SnapShot of the successor
        instruction to 1.
Performs steps noted below for successor instruction
        handling.   .   .  array.
}
If this is the first time the SnapShot for a successor instruction
has been visited
{
Copy the stack counter, virtual stack, virtual register array
        and jsr bit vector array into the SnapShot for the
        successor instruction
Set the changed bit for the successor instruction
}
Else      /* the instruction has been visited before */
{
If the stack counter in the Status Array does not match the
        stack counter in the existing SnapShot, or the two
        stacks are not identical with regard to data types
```

```
        (except for differently typed object handles)
        {
        Set VerificationSuccess to False
        Return with Abort return code value
        }
Merge the Virtual Stack and Virtual Register Array values
        into the values of the existing SnapShot:
        (A) if two corresponding stack elements or two
        corresponding register elements contain different
        object handles, replace the specified data type for the
        stack or register element with the closest common
        ancestor of the two handle types;
        (B) if two corresponding register.  .  .  instruction.
        /*  Note that return, break and continue instructions
            inside a code block protected by a "finally"
            exception handler are treated the same as a "jsr"
            instruction (for a subroutine call to the "finally"
            exception handler) for verification purposes. */
        Copy the jsr bit vectors into the SnapShot of the
          successor instructions only.  .  .  as the current instruction.
        Set the changed bit for each successor instruction for
          which the merging of the stack and register values
          caused any change to the successor instruction's
          SnapShot.
        }
}         /* End of Do.  .  .
```

=> d l7 1-6

1. 5,937,193, Aug. 10, 1999, Circuit arrangement for translating platform-independent instructions for execution on a hardware platform and method thereof; David Ross Evoy, 395/705 [IMAGE AVAILABLE]

2. 5,925,123, Jul. 20, 1999, Processor for executing instruction sets received from a network or from a local memory; Marc Tremblay, et al., 712/212, 23 [IMAGE AVAILABLE]

3. 5,923,892, Jul. 13, 1999, Host processor and coprocessor arrangement for processing platform-independent code; Paul S. Levy, 712/31; 709/208; 712/34, 36 [IMAGE AVAILABLE]

4. 5,884,083, Mar. 16, 1999, Computer system to compile non-incremental computer source code to execute within an incremental type computer system; Robert Royce, et al., 395/705, 707 [IMAGE AVAILABLE]

5. 5,784,553, Jul. 21, 1998, Method and system for generating a computer program test suite using dynamic symbolic execution of JAVA programs; Adam K. Kolawa, et al., 714/38 [IMAGE AVAILABLE]

6. 5,740,441, Apr. 14, 1998, Bytecode program interpreter apparatus and method with pre-verification of data type restrictions and object initialization; Frank Yellin, et al., 395/704, 705, 707 [IMAGE AVAILABLE]

```
L4          0 S L3 AND JAVA?/AB
L5          0 S L3 AND JAVA?
L6         47 S (EXECUTION (W) STACK?)
L7          1 S L6 AND JAVA?
```

(execution stack) (exception or interrupt)

```
=> s l6 and java?

          1161 JAVA?
L7           1 L6 AND JAVA?

=> d l7 kwic
```

US PAT NO: 5,933,635 [IMAGE AVAILABLE]   L7: 1 of 1

SUMMARY:

BSUM(2)

This . . . and Apparatus for Performing Byte-Code Optimization During Pauses, U.S. patent application Ser. No. 08/944,335 (Atty. Docket No. SUN1P150/P2300), entitled "Mixed **Execution Stack** and Exception Handling," U.S. patent application Ser. No. 08/944,326 (Atty: Docket No. SUN1P152/P2302), entitled "Method and Apparatus for Implementing Multiple. . .

DETDESC:

DETD(33)

As . . . 7, and is suitable for implementing the present invention. When a computer program, e.g., a computer program written in the **Java**.TM. programming language developed by Sun Microsystems of Mountain View, Calif., is executed, source code 810 is provided to a compiler. . .

DETDESC:

DETD(34)

Byte . . . executed on substantially any computer system that is running a suitable virtual machine 840. By way of example, in a **Java**.TM. environment, byte codes 830 may be executed on a computer system that is running a **Java**.TM. virtual machine.

DETDESC:

DETD(37)

When . . . In general, the machine-language instructions are discarded when virtual machine 840 terminates. The operation of virtual machines or, more particularly, **Java**.TM. virtual machines, is described in more detail in The **Java**.TM. Virtual Machine Specification by Tim Lindholm and Frank Yellin (ISBN 0-201-63452-X), which is incorporated herein by reference in its entirety.

```
=> d his

     (FILE 'USPAT' ENTERED AT 14:54:21 ON 02 SEP 1999)
L1          5147 S 712/?/CCLS
L2             0 S L1 AND (JAVA (P) (EXECUTION (W) STACK?))
L3             5 S L1 AND (EXECUTION (W) STACK?)
```

```
=> d his

     (FILE 'USPAT' ENTERED AT 16:13:11 ON 02 SEP 1999)
L1          5147 S 712/?/CCLS
L2            47 S (EXECUTION (W) STACK?)
L3          2779 S L1 AND (EXCEPTION? OR INTERRUPT?)
L4            11 S ((EXECUTION (W) STACK?) (P) (EXCEPTION? OR INTERRUPT?))

=> s l4 and java?

          1161 JAVA?
L5            1 L4 AND JAVA?

=> d l5
```

1. 5,933,635, Aug. 3, 1999, Method and apparatus for dynamically deoptimizing compiled activations; Urs Holzle, et al., 395/701 [IMAGE AVAILABLE]

```
=> d l4 1-11
```

1. 5,933,635, Aug. 3, 1999, Method and apparatus for dynamically deoptimizing compiled activations; Urs Holzle, et al., 395/701 [IMAGE AVAILABLE]

2. 5,590,332, Dec. 31, 1996, Garbage collection, tail recursion and first-class continuations in stack-oriented languages; Henry G. Baker, 395/705; 707/206 [IMAGE AVAILABLE]

3. 5,530,870, Jun. 25, 1996, Arrangement for efficiently transferring program execution between subprograms; Dennis L. De Bruler, 395/706; 364/261.3, 280.4, DIG.1 [IMAGE AVAILABLE]

4. 5,522,072, May 28, 1996, Arrangement for efficiently transferring program execution between subprograms; Dennis L. De Bruler, 709/304; 364/247.7, 265.6, 281.6, 281.7, 944.6, 957.6, 965.4, 967.4, DIG.1, DIG.2 [IMAGE AVAILABLE]

5. 5,412,717, May 2, 1995, Computer system security method and apparatus having program authorization information data structures; Addison M. Fischer, 380/4, 23, 25 [IMAGE AVAILABLE]

6. 5,311,591, May 10, 1994, Computer system security method and apparatus for creating and using program authorization information data structures; Addison M. Fischer, 380/4 [IMAGE AVAILABLE]

7. 5,109,329, Apr. 28, 1992, Multiprocessing method and arrangement; Brian K. Strelioff, 710/261; 364/230.2, 230.4, 232.9 [IMAGE AVAILABLE]

8. 5,003,466, Mar. 26, 1991, Multiprocessing method and arrangement; Edward P. Schan, Jr., et al., 714/41; 364/228.3, 230, 230.2, 230.4, 232.9, 238.3, 238.4, 240, 240.1, 241.9, 247, 247.7, 251, 251.3, 262.4, 264.5, 265, 265.3, 265.6, 266.6, 280, 280.2, DIG.1 [IMAGE AVAILABLE]

9. 4,597,044, Jun. 24, 1986, Apparatus and method for providing a

composite descriptor in a data processing system; Joseph C. Circello, 713/100; 364/231.8, 238, 239, 239.3, 239.4, 243, 243.4, 243.41, 243.42, 243.44, 244, 244.3, 263.1, 280, DIG.1; 711/200; 712/214; 713/200 [IMAGE AVAILABLE]

10. 4,530,052, Jul. 16, 1985, Apparatus and method for a data processing unit sharing a plurality of operating systems; James L. King, et al., 713/100; 364/241.7, 241.9, 243.41, 243.42, 243.44, 243.45, 256.3, 256.4, 265, 266.4, 268, 268.3, 268.5, 280, 280.2, DIG.1; 395/406R; 709/100; 711/200, 208 [IMAGE AVAILABLE]

11. 4,521,851, Jun. 4, 1985, Central processor; Leonard G. Trubisky, et al., 712/218; 364/222.5, 230, 230.3, 231.8, 231.9, 243, 243.4, 243.41, 243.42, 243.44, 244, 244.3, 247, 247.6, 252, 256.3, 259, 259.2, 263, 263.1, 286, 286.4, DIG.1; 712/23 [IMAGE AVAILABLE]